Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра інформатики факультету інформатики

Deep Learning-based military vehicle recognition in UAV video footage

Текстова частина до курсової роботи за спеціальністю «Комп'ютерні Науки»

Керівник курсової роботи

ст. викладач Кузьменко Д. О.

_____

(підпис)

"____"_____2024 р.

Виконав студент 3 р. н.

Безбородов В. П.

"____"_____2024 р.

Календарний план виконання курсової роботи

№ п/п	Назва етапу дипломного проекту (роботи)	Термін виконання етапу	Примітка
1.	Отримання завдання на курсову роботу	Жовтень 2023 р.	
2.	Аналіз предметної області	Листопад 2023 р.	
3.	Опрацювання літератури	Січень-лютий 2024 р.	
4.	Виконання практичної частини	Лютий-Квітень 2024 р.	
5.	Написання текстової частини курсової	Травень 2024 р.	
6.	Оформлення презентації для захисту	Травень 2024 р.	
7.	Захист курсової роботи	22 травня 2024 р.	

Студент: Безбородов В. П.

Керівник: Кузьменко Д. О.

Table of Contents

1 ABSTRACT 4
2 INTRODUCTION 5
3 THEORETICAL FRAMEWORKS 8
3.1 Introduction to object detection 8
3.2 YOLO 13
3.3 YOLO Mechanism 14
3.4 YOLOv5 architecture 15
3.5 YOLOv8 architecture 18
3.6 Model optimizations 19
3.7 Model efficiency 20
4 EXPERIMENTAL RESEARCH 23
4.1 Dataset and models set 23
4.2 Experiments 24
5 RESULTS 25
6 CONCLUSIONS 35
6.1 Future scope 35
REFERENCES 36
APPENDIX 1 40

1 ABSTRACT

The course work addresses the creation of neural network models for detecting military vehicles from unmanned aerial vehicle (UAV) images using deep learning. The work describes the analysis and comparison of YOLO models, and the search for the optimal approach in terms of size and accuracy of military vehicle detection. The solution is planned to be embedded into the hardware UAVs in the future.

Keywords: Python, PyTorch, Deep Learning, Neural Network, YOLOv8, YOLOv5

2 INTRODUCTION

Drones, also known as uncrewed aerial vehicles (UAV) [1] were first developed in the twentieth century by the military sector in order to perform dangerous tasks for humans. Throughout decades drones have become more efficient, capable of traversing longer distances, and less costly in terms of money and production time, which has resulted in this technology being expanded in different areas and,
where drones' advantages prevail. In this work, we will discuss the use of drones in the military domain. Owing to modern technological advancements, drones have become an indispensable tool in the miltech. There is a wide variety of drones types, divided into different categories: by size, maximum take-off weight, operating height and
speed. Furthermore, it is possible to modify commercial drones for military use. For example, it is a combination of Miltech and the Armed Forces of Ukraine. Applying and improving such ranges of models as DJI Matrice 300, DJI Mavic 3, DJI Mavic 2, Autel Light +, Autel Evo 2 Pro, and Autel Evo 2 Pro Enterprise [2] the military gains an additional advantage in the air. These models are usually used in the following combat operations:

- reconnaissance, the operations where a drone can either patrol the designated territory or fly within a certain distance of the enemy's territory, which is limited either by the UAV's capacity or by the task scope.

- attacking targets, the operations where the vehicle tries to quickly fly to a given point, drop a combat charge, such as a grenade, mortar shell, or other explosive devices, hit the target, and return and change location just as quickly to avoid drone operators from tracking enemy attacks.

- support missions, the operations where the UAV helps allied units to adjust fire on targets by providing certain information, adjusting artillery fire, or supporting infantry units from the sky.

During each of the above tasks, the UAV's flight is recorded for later analysis, since during the operation the drone can capture military equipment, enemy positions, etc. After receiving the video data, it gets scrupulously inspected to detect objects and consequently passes the information on. The biggest problem at this stage is that, until recently, it was done manually which is difficult and very time-consuming. Therefore, on the miltech side, a solution to this problem has emerged through the use of deep learning and object detection by neural networks, thereby significantly speeding up the processing of video recordings.

This research aims to investigate object detection models, compare them to select the one with the highest accuracy, and try to optimize their capacity and integrate the deep learning model in embedded hardware used by UAVs. One of the possible options is NVIDIA Jetson Nano [3] which can infer neural networks efficiently on embedded devices. We outlined the following steps for this work:

1. Look for an appropriate dataset.

2. Explore and become familiar with object detection models.

3. Train and optimize models using the retrieved dataset.

4. Analyse and compare the results.

There are many different opensource object detection models, whose object detection algorithms have different ways of image feature extraction. One popular architecture is YOLO(You Only Look Once). These are mainstream models that work efficiently by maintaining a good balance between model size, accuracy, and inference speed. There are many recent works where YOLO models have contributed to object detection from UAV scenarios that are similar to our research. Wang G. and Chen Y. in their work [4] offered an improved custom YOLOv8 model for UAV Aerial Photography Scenarios. Zhai X., et al. [5] suggested an optimized YOLOv8, but their goal was to use the neural network to detect tiny UAVs. Another example provides Li Y., et al. [6] who implement a modified YOLOv8 detection network for UAV aerial image recognition. In

addition, there is research that has been conducted using other YOLO models like YOLOv5 and others. At the current time, the latest model is YOLOv9 which outperforms previous generations, but as it is a relatively new model and it might have an inconsistent behaviour during training, while YOLOv8, a predecessor, is more stable.

For our experiments, we will be use the YOLOv5 and YOLOv8 models to explore best neutral architecture for our tasks. The following work is be separated into sections each describing a significant part of deep learning training.

Section 3 describes the theoretical part of our research and explores YOLO models in detail – the intricacies of their training, hyperparameter tuning and the intermittent results.

Section 4 introduces the experimental part of the work where we discuss our ways of optimizing for good precision on the chosen dataset.

Section 5 analyses the results of our trained neural networks. The conclusions are outlined in Section 6.

3 THEORETICAL FRAMEWORKS

3.1 Introduction to object detection

Object detection is a staple task related to computer vision and image processing. It detects examples of objects of a certain class such as humans, buildings. In our work, the objects for detection are military vehicles in digital images and videos. It can be considered the union of two core tasks in computer vision: object localization and image classification. The former refers to the ability to recognize multiple objects' positions in a single image. The second latter allows for the identification of the class label for these objects. When we put them together, we can find the location, regress bounding boxes, the rectangle-shaped contours of the object frame, and identify what type of object it is.

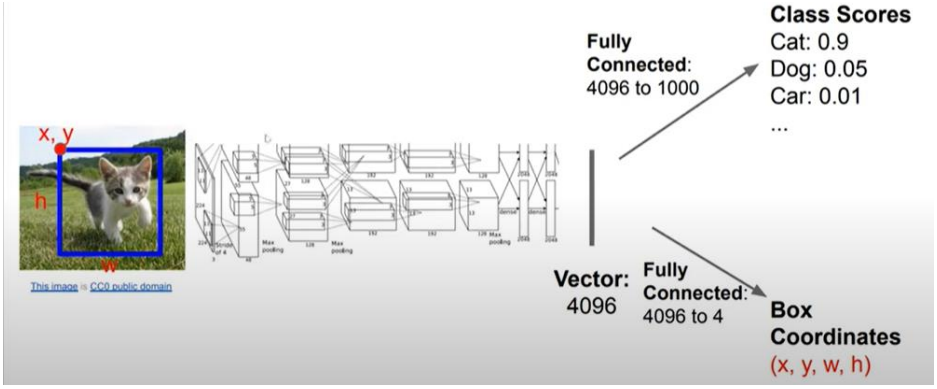

Figure 1. Example of a schema detection of the single object

In Figure 1 there is a standard input, a rectangle with x, and y coordinates, top left corner, width, and height (bounding box). The idea is to extract features from the image via Convolutional Neural Network (CNN) [7] and perform classification and localization. First, find the objects, and then classifying them. Then we find the area of interest and simply perform image classification. The arrow below in Figure 1 points to the bounding box. This is the task of regression of the bounding box that requires us to select the coordinates [*x*, *y*, *w*, *h*] to capture the object as precisely as possible.

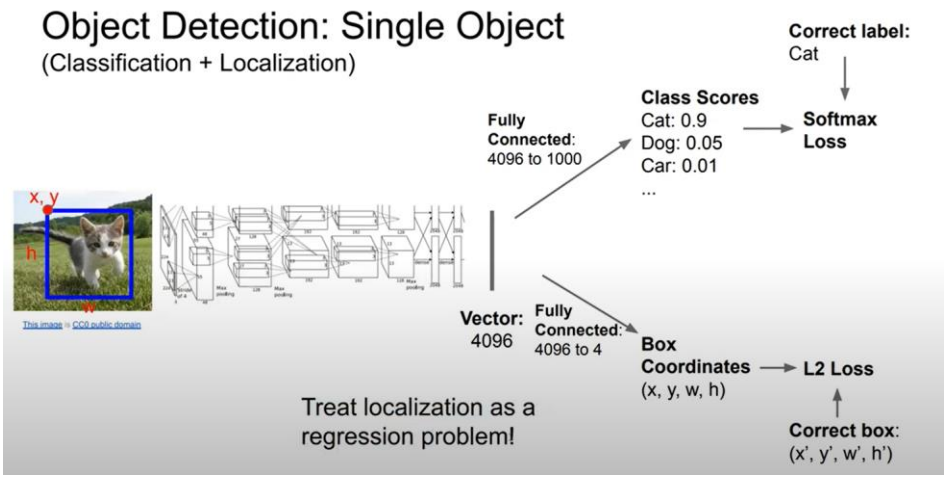

Figure 2. Further explanation of the instance

We treat localization as a regression problem. We calculate L2 loss [8] with the predicted coordinates and the ground truth, as depicted in Figure 2.

L2 Loss is a commonly used function in regression tasks. It measures a squared difference between the predicted and true values in the dataset. The higher the deviation from the true value is, the bigger the penalty applied.

$L_2(x)=x^2$

$$f(y\_predicted, y\_true) = \sum_{i=1}^{n} (y\_predicted - y\_true)\text{^}2$$

SoftMax Loss [9] returns class scores for each class label. It is commonly the last layer at the end of the neural network. The final loss is the combinations of the two losses to be optimized.

If multiple objects are to be detected, another problem arises: how to assign potential areas of interest, which may contain such objects. Figure 1 demonstrates a single object of interest (cat) in the plain view. However, it is much more complicated to detect multiple objects of different shapes, scales, sizes, and occlusion. The first idea suggests applying a CNN to many different crops of the image: CNN classifies each crop as an object or background. The problem with this approach is that is very time-consuming and computationally expensive, as it needs to apply a CNN to a huge number of locations,

scales, and aspect ratios. Research by Ross Girshick, et al. [10] suggested a new approach Region Proposals: Selective Search and CNN (R-CNN). The main idea lies within generating regions that are likely to have objects and processing them with a CNN to extract features. The advantage is that it is a faster method than before, as it gives 2000 region proposals in a few seconds on CPU.

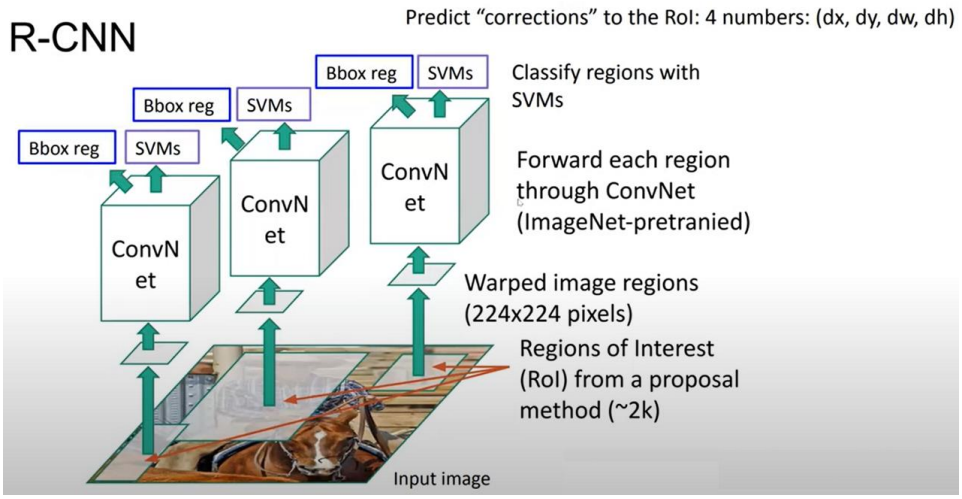

Figure 3. Rundown of the R-CNN object detection system

As Figure 3 illustrates, there is an image as input, at first, we take the generated regions of interest (ROI), resize them, and then infer each of them through with a pretrained CNN.Then, classification is performed. In the example of Figure 3, the classification is done using SVM and then providing bounding boxes. The problem with the model is that the algorithm is still slow. It needs to process 2K independent forward passes for each image. One possible solution was to pass the image through CNN before cropping, so cropping the convolutional features is suggested instead.

Later Ross Girschick produced a Fast R-CNN [11] as an improvement of the previous model.

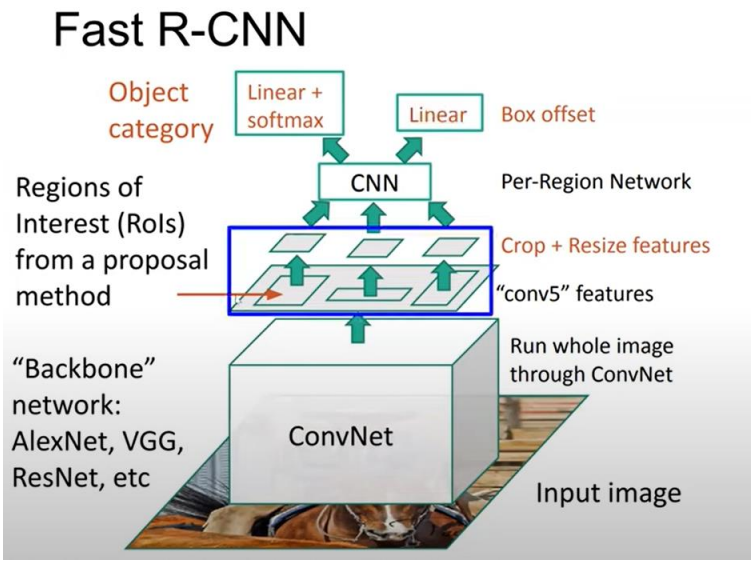

Figure 4. Overview of the Fast R-CNN system

As depicted in Figure 4, at the beginning we collected all the features from the image, then generated regions of interest or RoIs from them. After that, the authors resampled and cropped them to a smaller image dimension, and finally ran them through CNN and calculated the classification and bounding boxes. The key point is highlighted in blue, which significantly improved the model, by introducing the RoI Pooling layer. The solution implements max pooling for each region converting features into a small features maps. Thereby, collecting the most important information and resizing the image. Even though this model completed the goal and was significantly faster, the problem of time-consuming region proposals remained.

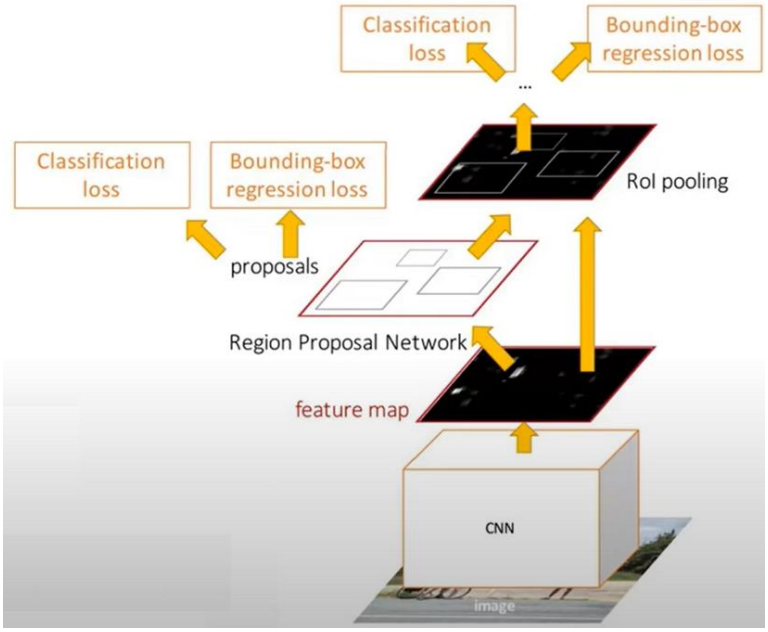

Figure 5. Faster R-CNN system

To solve the mentioned disadvantage, a Faster R-CNN[12] was developed where CNN was used as a tool for generating proposals. The authors separately identified the RoI generation module in the Region Proposal Network (RPN) to predict proposals from features. After this layer, they additionally defined the classification loss and bounding-box regression loss as an additional verification, and concluded with RoI Pooling same losses.

RPN works as follows: it introduced a notion of a fixed-size anchor field at each point in the feature map - the center point of a large kernel that is glued to a pixel in the image. There are many such anchors. At each point, the authors predict whether the corresponding anchor contains an object. For certain boxes, they also predict the corrections from the anchor to the true box by regressing 4 numbers *(x, y, w, h)* on the pixel. In practice, they use *K* different anchor boxes of different scales at each location. If the anchor is an object, the box is transformed using the following algorithm: Sort *K*w*h* boxes according to how well they detect objects, and select the top 300 as proposals from those.

There are two main types of object detection architectures: Single-Stage Object Detectors and Two-Stage Object Detectors. The difference between them is an approach to

generating region proposals and implementing object classification. Faster R-CNN is a Two-Stage detector, as it explicitly generates region proposals as a separate step before classification, while, in contrast, a One-Stage detector works in one pass directly predicting bounding boxes. Therefore, Single-Stage is generally faster, but Two-Stage is considered more accurate.

3.2 YOLO

"You Only Look Once: Unified, Real-Time Object Detection" [13] was first introduced in 2016 by Joseph Redmon et al. They devised a method for a single neural network to predict bounding boxes and infer class probabilities for objects on full images in a single pass. Even though the first model had some limitations, it still was a breakthrough and brought much potential. Moving forward, in 2017, Joseph Redmon and Ali Farhadi introduced the next model YOLOv2, also called YOLO9000, [14] with made optimizations and enhancements. The YOLOv3 [15] model was presented in 2018, making it more complex and accurate. As the YOLO family models were being adapted, over time other researchers contributed to further development. In 2020 Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao launched a YOLOv4 [16] model as a continuation of the YOLO family, right after 50 days Glenn Jocher published the YOLOv5 [17] source code as a YOLOv3 upgrade with plenty of extensions that outformed its predecessor in terms of optimization and performance changes. The main difference is taking the YOLO architecture and then porting it to a different framework, namely, PyTorch [18], possessing common improvements as YOLOv4. The latest models out of date are YOLOv8 [19] and YOLOv9 [20]. Both are open-source, fast and accurate.

3.3 YOLO Mechanism

The concept of implementation of Single-Stage Detectors is common to Two-Stage, but in this case, there are more anchor boxes, better optimized algorithms on how to choose them, architectures, etc.

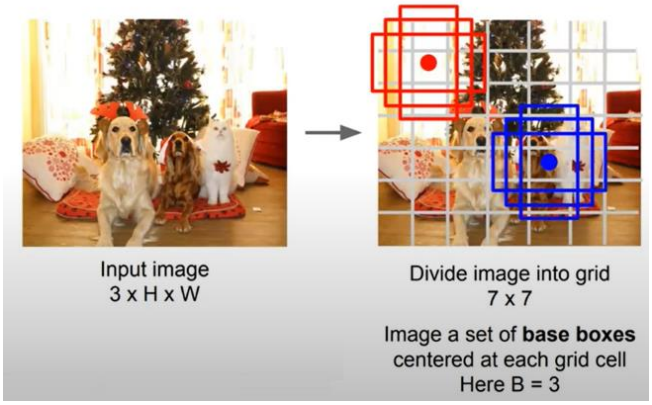

Figure 6. Instance of Single-Stage Detector in Practice

As Figure 6 shows, the image is divided into grids by *S***S* size and then within each grid cell next process takes place:

- Regressing from each of the B base boxes to a final box with 5 numbers: (*dx*, *dy*, *dw*, *dh*, *confidence)*, x,y – are the coordinates of the box center within a grid cell, width and height are relative to the entire image, confidence reflects the IoU(Intersection over Union) value between the predicted box and the ground truth. IoU measures the overlap between two bounding boxes by computing the ratio of their intersection to their union. This metric assists assess the accuracy of the object detection by indicating how well the predicted box aligns with the actual object.

- Predicting scores for each of *C* classes including background as a class

After these steps, a Non-Max Suppression (NMS) [] is used. It is designated as reducing the number of overlapping bounding boxes, therefore NMS removes duplicate

object detections and selects the most relevant one. This helps reduce computational complexity.

3.4 YOLOv5 architecture

In broad terms, the YOLO architecture can be separated into three main parts: the backbone, the neck, and the head. In the backbone, features get extracted from the image. The neck refines these features, improving spatial and semantic information. Finally, the head uses these refined properties to perform object detection. Diving into the architecture we will discuss the key modules and their purpose.

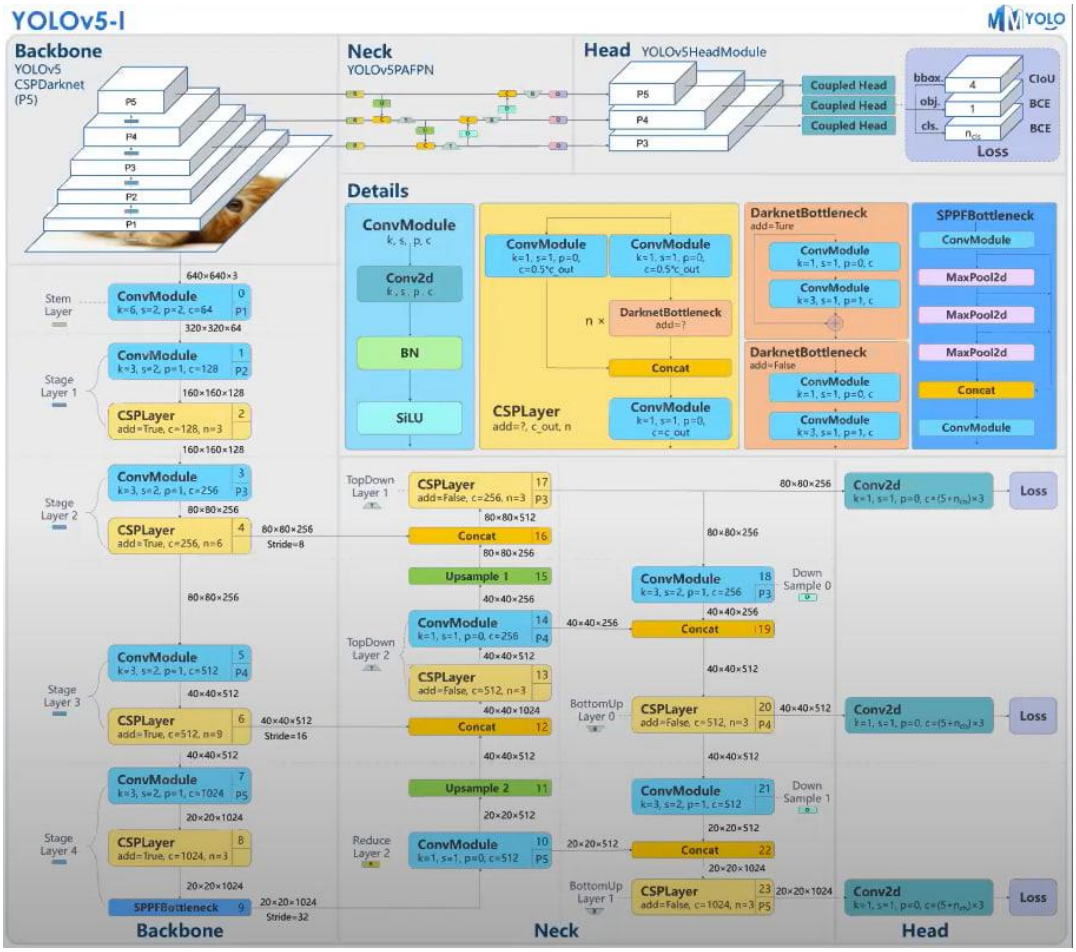

Figure 7. YOLOv5 Architecture [21]

In Figure 7 we can outline the Convolutional Module that appears in different stages of the architecture. Fundamentally, it consists of a Convolutional Layer which uses a convolutional kernel as it slides over the image spatially. Stride is responsible for how

many pixels the kernel can skip. These filters extend the full depth of the input volume. In addition, convolving padding is used for preserving input spatial dimensions in output activations. As the output the activation or feature map is obtained. Considering the number of the filters in the end we get the same number of the activation maps. The next step is Batch Normalization [22]. Sergey Ioffe and Christian Szegedy proposed the idea of normalizing the inputs in the following way.

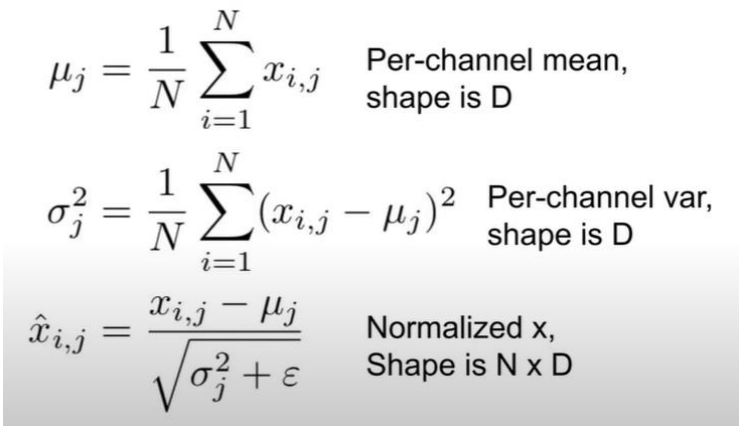

Figure 8. Step-by-step Batch Norm implementation

In Figure 8 there is a clear demonstration of Batch Norm where, for example, we have an input *x: N*D* size, then we can calculate the average per channel, sum our matrix values and divide by N, then calculate the channel variance, subtract the average from the value, square it, divide by N, and finally normalize it. By its mechanics Batch Norm stabilizes and accelerates the training of Neural Networks.

The last part of the Convolutional Module is the activation function.
Mainly they provide non-linearity that is important during the model training. This function zeroes negative weights and passes forward the rest. Other types of activation functions allow a small number of negative values to pass through so that the gradient does not get vanished. They are used to nonlinearly distribute data that is nonlinearly separable. The YOLO architecture uses SiLU function activation [23].

The CSP(Cross Stage Partial) Layer[24] helps reduce the computational cost while maintaining the learning capacity. The interaction between BottleneckCSP and regular

convolutional layers is that the BottleneckCSP layers are used to enhance feature extraction and learning efficiency, while the convolutional layers perform the standard convolution operations.

All in all, we have many Convolutional Modules and CSP layers. Next important stage during model training is the Max Pooling layer that is in SPPFBottleneck. It is the operation of reducing the image dimension, usually in half, to simplify the testing of the features by a neural network and simplify calculations. The smaller the image, the easier it is for the hardware to process it. Pooling calculates the pixel with the maximum value within the area where the filter is, sliding over the feature map a pooled or down-sampled feature map.

In the neck stage further refinements of the extracted features take place before passing them to the head. In the middle, the idea of up-sampling or “unpooling” is used. Generally, we create new feature maps with more dimensions from the fused ones. There are some approaches to implement this such as “Nearest Neighbor” [25] and “Bed of Nails” [25].

The final output is computed in the head layer and is used to compute the cross entropy for classification, the L1 bounding box positions, and the objectness loss. In Figure 7, the innings of the YOLO model are displayed.

YOLOv5 is available in different versions [17] where parameters such as the width and depth of convolutional modules vary to perform them in tasks.

3.5 YOLOv8 architecture

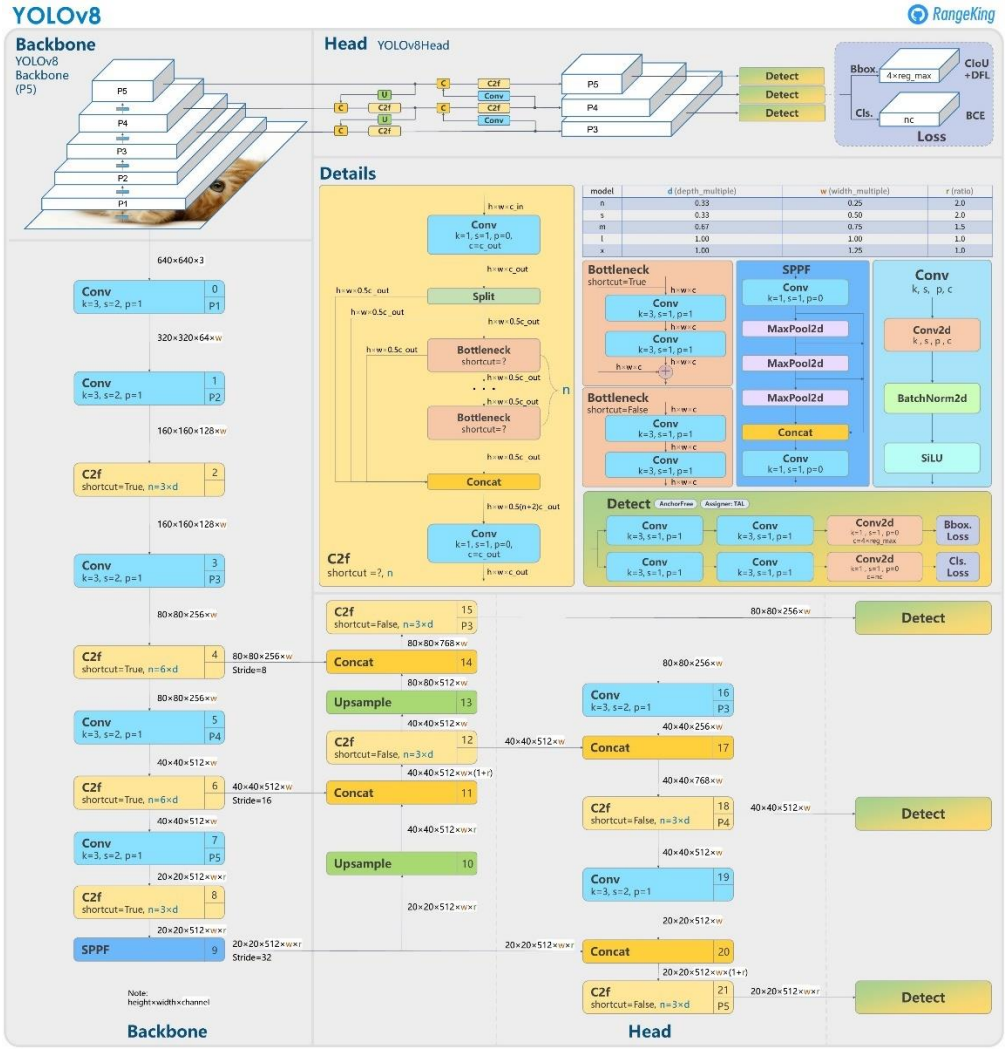

Figure 9. YOLOv8 architecture [26]

Comparing the architectures from Figure 7 and Figure 9, it is noticeable notice that there are many common structures. The novelty of the YOLOv8 model can be explained in the following summary [27]:

- Providing new C2f module instead of C3 or CSP layer.

- Changing the first Convolutional Module 6 by 6 size to Convolutional Module 3 by 3 size in the backbone.

- Replacement of the first Convolutional Module 1 by 1 size with Convolutional Module 3 by 3 size in the bottleneck.

- Deleting two Convolutional Modules (№10 and №14 in YOLOv5 config)

- Using a decoupled head and removing the objectness branch.

As an additional improvement, a detection without manual anchor box assignment was introduced. YOLOv8 can automatically detect the central point of an object rather than the object's offset from the center of the pre-defined anchor box of that region. Anchor-free detection reduces the number of box predictions, which speeds up NMS. This is presented as a significant alteration to the connection head.

YOLOv8 provides different versions[19] as well the ones suited for specific applications.

3.6 Model optimizations

Researchers have noticed that different model architectures can impact performance. By experimenting with various structures, they discovered ways to optimize models, which can slightly improve their speed or accuracy. These small advancements are possible to achieve by using gradient descent in the model's training. In general, as inputs it gets a loss function, data, and weights and updates the following way. First, it calculates the gradient's value, then according to a learning rate or a step size [28] updates the weights and reaches a convergence point where the optimal number of parameters is found. There are many such optimizers but in this research, only Adam [29] and SGD [29] will be outlined.

The idea that lies behind Stochastic gradient descent is using mini-batching, calculating the gradient of the loss function with 32 or 64 or another number that is the power of 2 at each iteration. Not much different from the standard one, it decreases computational complexity.

Adam is an acronym for Adaptive moment and uses an adaptive learning rate for estimation of the mean and gradient's variance to adaptively scale the learning rate during the training that might improve the model's performance.

The important hyperparameters for these gradients are:

- Learning rate determines the size of steps we take to reach a local minimum. Making it larger or smaller can affect the number of iterations leading to convergence. During training the learning rate can become obsolete, and it needs to be updated. For this occasion, the definition of learning rate scheduling is introduced. It helps refresh the value of the learning rate, so it could break out of the local minimum and reach a global minimum. There are many variations, and functions, with which one can update the value such as Cosine, Linear, Inverse sqrt, etc.

- Batch size fixates several samples used to compute the gradient at each iteration which also complicates the search for optimal hyperparameters.

- SGD with momentum can play a crucial role during training as it determines a number of previous updates to the current one, helping the algorithm overcome a shallow local minimum.

3.7 Model efficiency

When we train object detection models, we operate with certain metrics to evaluate their performance. For object detection two common metrics are precision and recall [30]: the number of correctly detected objects within all detected objects, and the number of the correctly detected objects within all relevant objects, respectively.

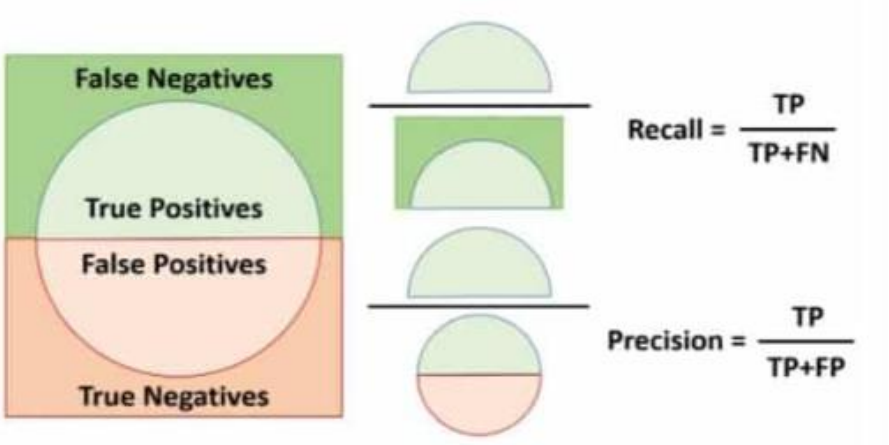

Figure 10. Visualization of recall and precision calculations

As Figure 10 illustrates, the definitions are based on two different types of error:

- Type I error or False Positives, objects that were detected but are incorrect.
- Type II error or False Negatives, the model failed to detect the correct objects.

Based on these metrics, we can evaluate the performance of a model. Out of precision and recall, we can calculate the Average Precision or AP, the area under the precision-recall curve to provide a more accurate score.

Mean Average Precision or mAP extends the concept of AP by averaging AP values across the object classes in the training sample as it is depicted in Figure 11.

Mean Average Precision (mAP)
Using the COCO Evaluator

$$mAP = \frac{1}{|classes|} \sum_{c \in classes} \frac{|TP_c|}{|FP_c| + |TP_c|}$$

Figure 11. Computing of mean Average Precision

Furthermore, there is another concept that impacts the values of mAP, namely Intersection over Union or IoU.

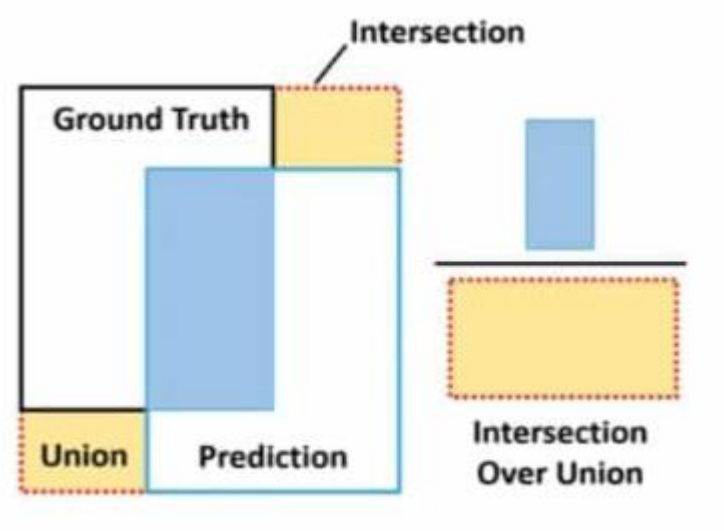

Figure 12. Notion of Intersection Over Union

This metric sets a threshold for a ratio of predicted bounding boxes to ground truth ones. And by these criteria, we get a range of mAPs. The threshold is usually set between 0.5 and 0.95 (mAP50, mAP50-95)

4 EXPERIMENTAL RESEARCH

4.1 Dataset and models set

For the practical survey we chose 2 main dataset sources: Kaggle [31] and Roboflow Universe [32]. We found an appropriate dataset containing images of the military vehicles [33]. There is a newer version of the dataset as of now, but we used the first version during the experiments. All trials were conducted locally in PyCharm IDE using GPU NVIDIA GeForce GTX 1070 8192MiB.

The dataset consists of 6515 images and is split into three parts: train, validation, and test, each containing 6186, 214, 115 image, respectively. There are 7 classes:

- AA – Anti-Aircraft
- HAW - Heavy Armored Wheeled
- HT – Heavy Transport
- LAW – Light Armored Wheeled
- MAT - Medium Armored Tracked
- MRAP - Mine-Resistant Ambush Protected
- MT – Medium Transport

We focused on models with a good balance of accuracy and speed. We chose YOLO family for its high mAP and a superior performance in real time object detection. YOLOv5 and YOLOv8 were selected because they are available and opensource. Specifically, we used YOLOv5 and YOLOv8 because they are available open-source. As mentioned in the previous sections, these models have different versions. We used YOLOv5-nano and YOLOv8-nano as they have the least number of parameters and, therefore, are faster.

4.2 Experiments

Ultralytics YOLOv8-nano [19] trained on COCO [34] accomplishes 37.3 mAP and has 3.2 M parameters. We are aiming to reduce the model size so it could be integrated in Jetson Nano. There are certain parameters with which we can finetune to rescale the model. There are model compound scaling constants such as *depth, width and max_channels*. *Depth* parameter [35] determines how many bottleneck blocks are in C2f Module and the idea behind it lies in that deeper CNN can extract more complex features. *Width* [35] influences the ability precisely to capture divided features, increasing or decreasing the number of channels in convolutional layers. *Max_channels* [36] set the limit of usage of channels in the architecture preventing the model from becoming too wide. As described in Figure 9, YOLOv8 versions use different values for these three scales. For YOLOv8-nano the values of 0.33 for depth, 0.25 for width, and 1024 for max_channels are used. We tried different values for fusing the model, but settled on the following ones:

Depth: 0.25
Width: 0.25
Max_channels: 768 (1.5 ratio)

Based on this build we trained 16 models with different hyperparameters.

YOLOv5-nano has 1.9 M parameters [17] which is significantly smaller than YOLOv8-nano, so this model was trained without any changes in the architecture to reach the highest performance with the same hyperparameters. However, there was no dropout parameter as it was not implemented in the model. In the process of the experiments, we found out that there is an approach to modify architecture and apply a Dropout layer [37], but it was not done in the experimental part. Based on that 8 models were trained.

5 RESULTS

The results and hyperparameters of the trained models are displayed in Table 1.

Model#	Optimizer	Batch_Size	Cosine_LR	Dropout	Training_Time_in_minutes	metrics/precision (B)	metrics/recall (B)	metrics/mAP50 (B)	metrics/mAP 50-95(B)
0	Adam	16	False	0.00	61,8195	0.65896	0.62573	0.65688	0.44139
1	Adam	16	False	0.25	63.31476	0.65896	0.62573	0.65688	0.44139
2	Adam	16	True	0.00	60.50538	0.64966	0.58929	0.62591	0.42788
3	Adam	16	True	0.25	59.13336	0.64966	0.58929	0.62591	0.42788
4	Adam	32	False	0.00	43.43358	0.67982	0.58005	0.63276	0.43982
5	Adam	32	False	0.25	44.46096	0.67982	0.58005	0.63276	0.43982
6	Adam	32	True	0.00	45.64986	0.70473	0.60341	0.63550	0.44062
7	Adam	32	True	0.25	57.01296	0.70473	0.60341	0.63550	0.44062
8	SGD	16	False	0.00	74,05392	0.70702	0.60424	0.66581	**0.44564**
9	SGD	16	False	0.25	61.04646	0.70702	0.60424	0.66581	**0.44564**
10	SGD	16	True	0.00	59.2521	0.69789	0.61459	**0.67149**	0.43654
11	SGD	16	True	0.25	58.90374	0.69789	0.61459	**0.67149**	0.43654
12	SGD	32	False	0.00	**42.97128**	**0.72858**	0.52467	0.62019	0.41794
13	SGD	32	False	0.25	46.22964	**0.72858**	0.52467	0.62019	0.41794
14	SGD	32	True	0.00	50.4066	0.60772	**0.63352**	0.61230	0.40222
15	SGD	32	True	0.25	50.22804	0.60772	**0.63352**	0.61230	0.40222

Table 1. Results of custom YOLOv8 nano training

Among these indicators depicted in the Table 1, models 12 and 13 perform the highest precision. The resulting models with best precision-recall balance are models from 8 to 11. They show their superiority in comparison with others. We chose models 9 and 11 as the best among YOLOv8 models.

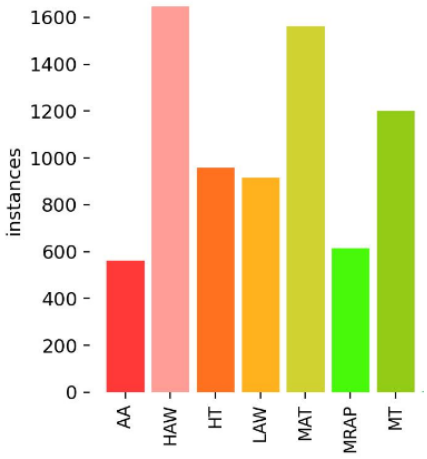

Figure 14. Distribution of the images across the classes

Figure 14 provides an insight into the distribution of the images between the classes in our dataset.

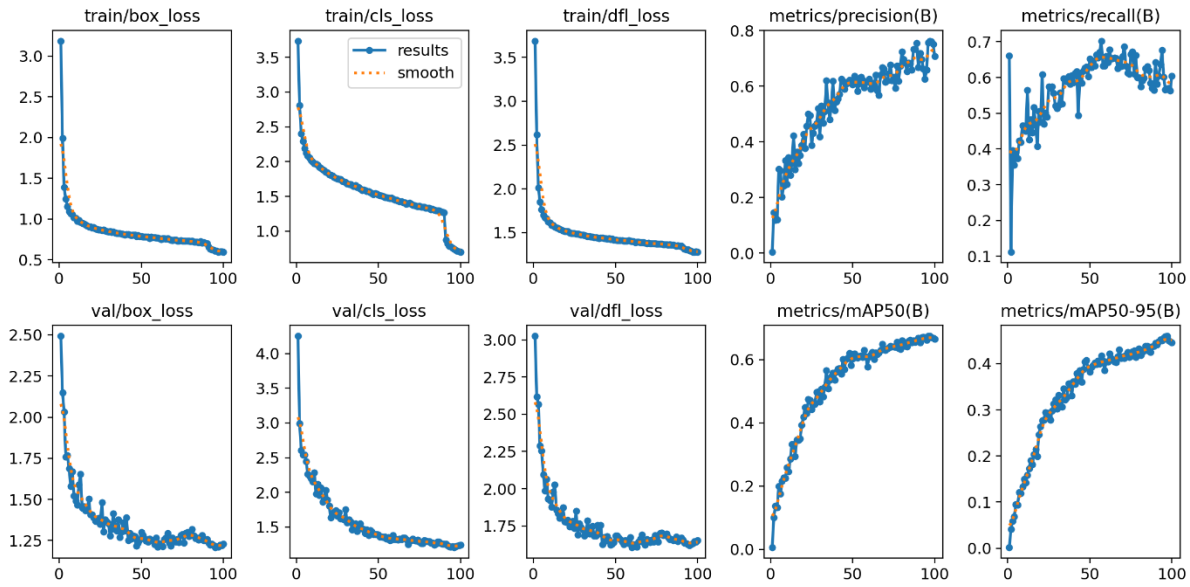

Figure 15. Result's training model 9.

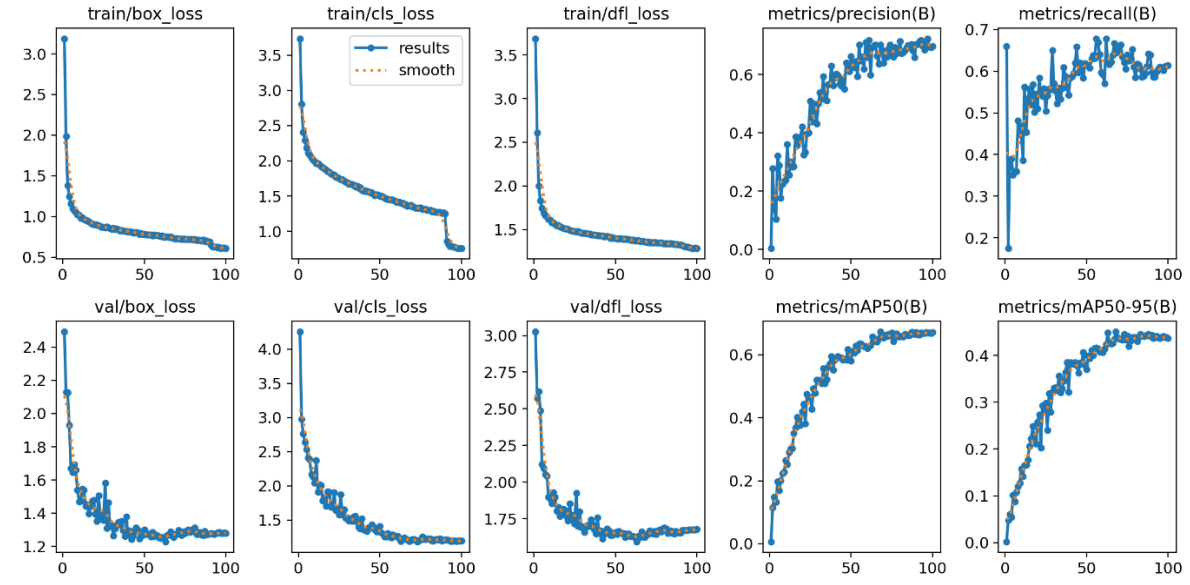

Figure 16. Result's training of model 11.

Figures 15 and 16 illustrate results visualizations of training of models 9 and 11across epochs, showing smooth and uninterrupted line in the loss functions, apart from training class loss function. The reason may be within the step of the learning rate that contributed to reducing the loss function indicator. They are not overfitting and are

showing good performance in each Figure. The recalls of both models reach their highest value at around epoch 60, then they recede. It may occur due to the enlarged threshold value, yielding fewer classified objects which can lead to the decreasing of the indicator. Simultaneously, the precision gradually becomes higher.

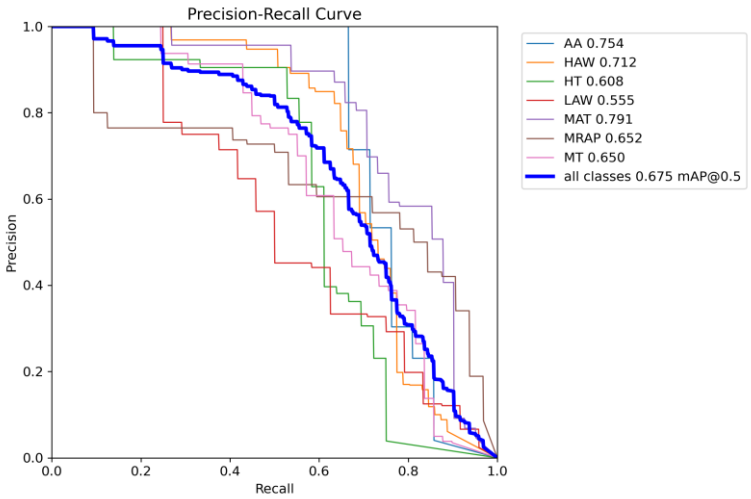

Figure 17. Precision-Recall Curve of model 9

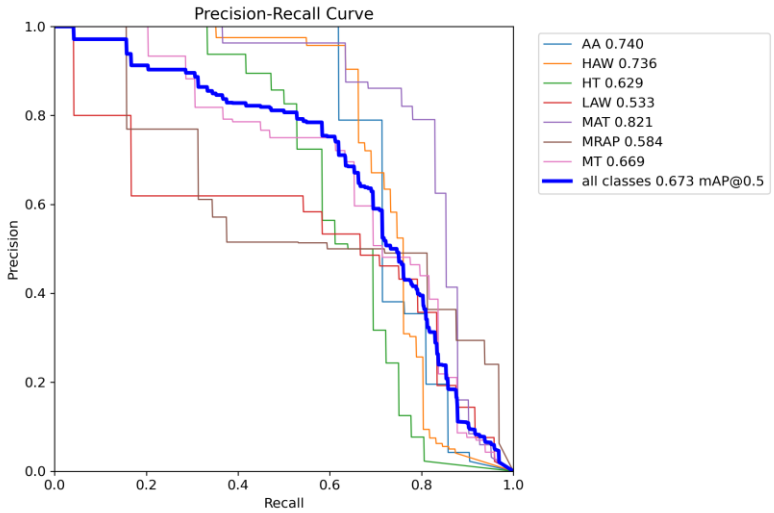

Figure 18. Precision-Recall Curve of model 11

Precision-recall curve shows the compromise between precision and recall for different thresholds. Comparing this proportion between two models, model 11 shows some looseness across the classes in contrast to model 9. However, both models provide similar mAP value ~0.67 that is a good indicator.

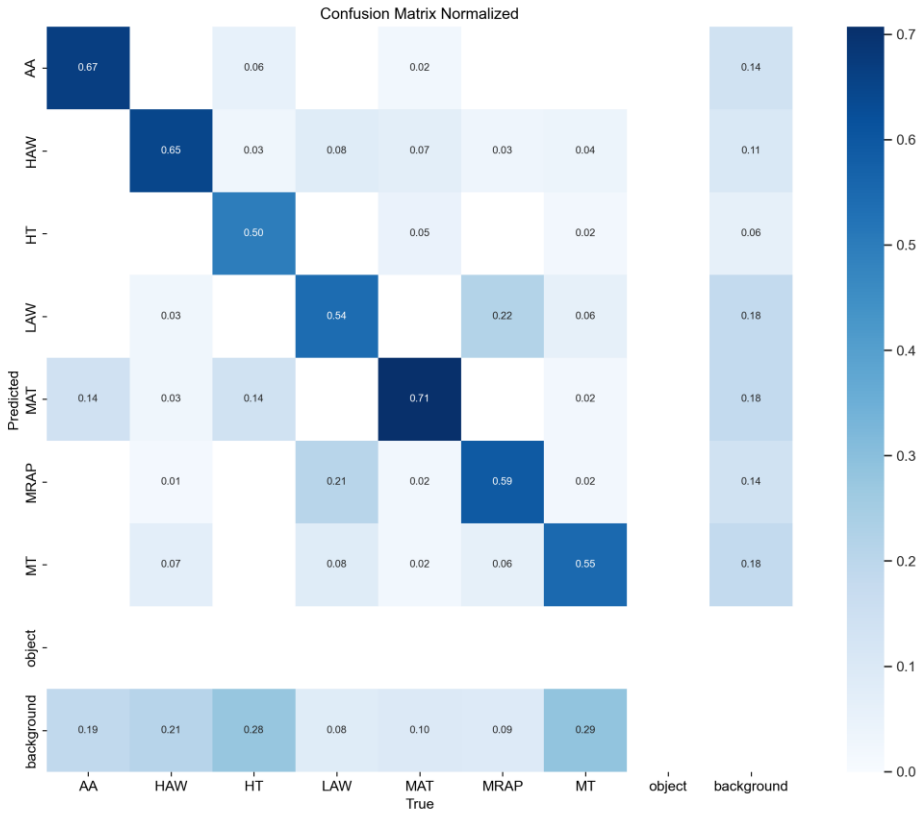

Figure 19. Normalized confusion matrix of model 9

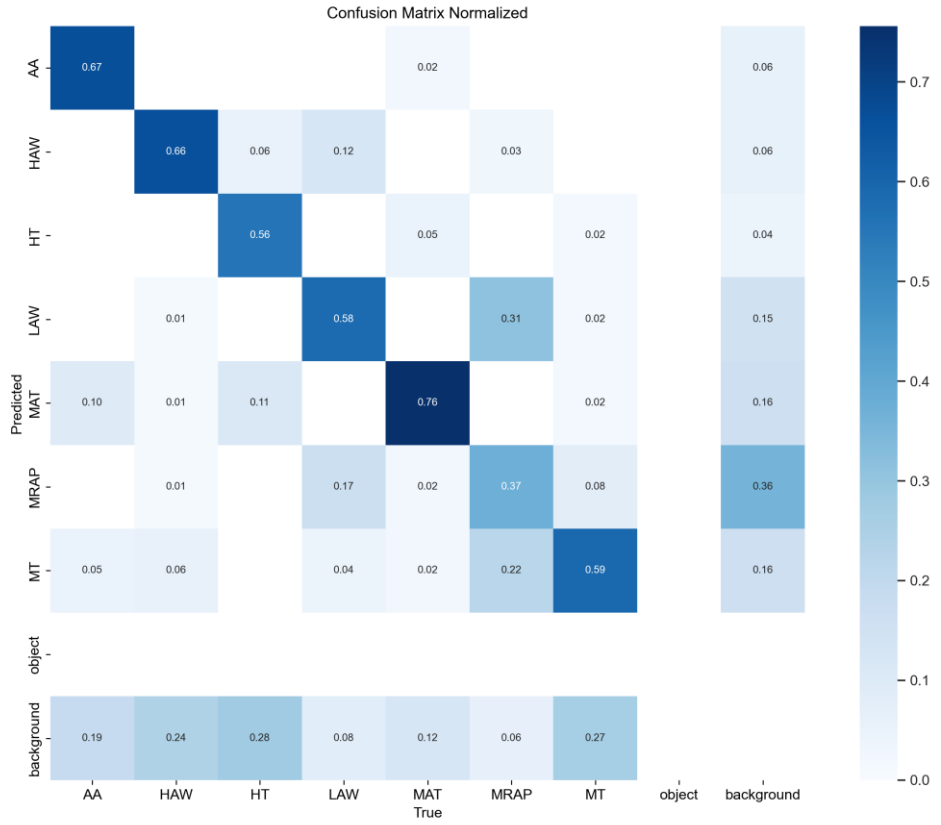

Figure 20. Normalized confusion matrix of model 11

Confusion matrix normalized also known as an error matrix (Figures 19 and 20) depicts a summary of predictions. It is intertwined with type I and II errors that we described in the previous section. Values along the main diagonal show True Positives (TP), correctly classified normalized values. The left of the diagonal demonstrates the False Positives (FP), objects that were classified incorrectly. The right of the diagonal provides the False Negatives (FN), objects that were present in the image but were not detected. The reason for these errors lies within the dataset. The model is trying to detect military vehicle that can often be camouflaged and the model can confuse the vehicle with part of natural background. Based on these assumptions and results we anticipate that model 11 misclassifies the military vehicle for natural background (or vice versa) more frequently, despite the fact that both models perform well.

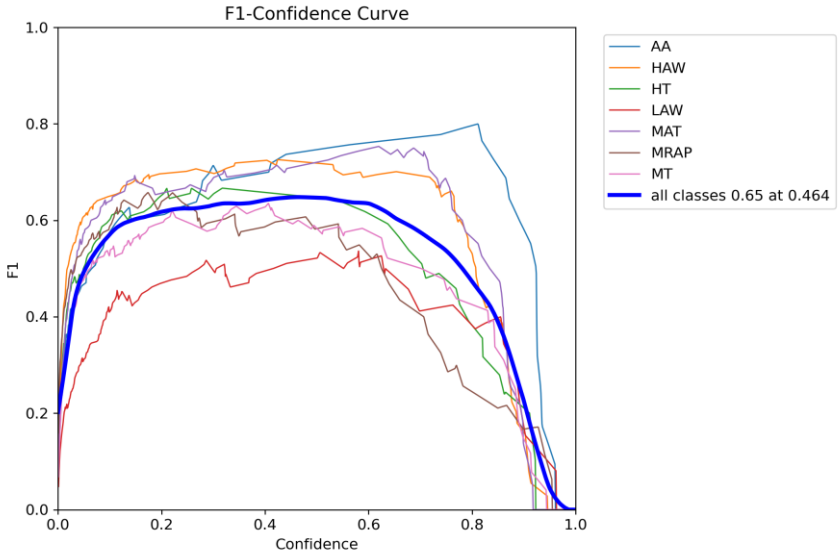

Figure 21. F1-Score of model 9

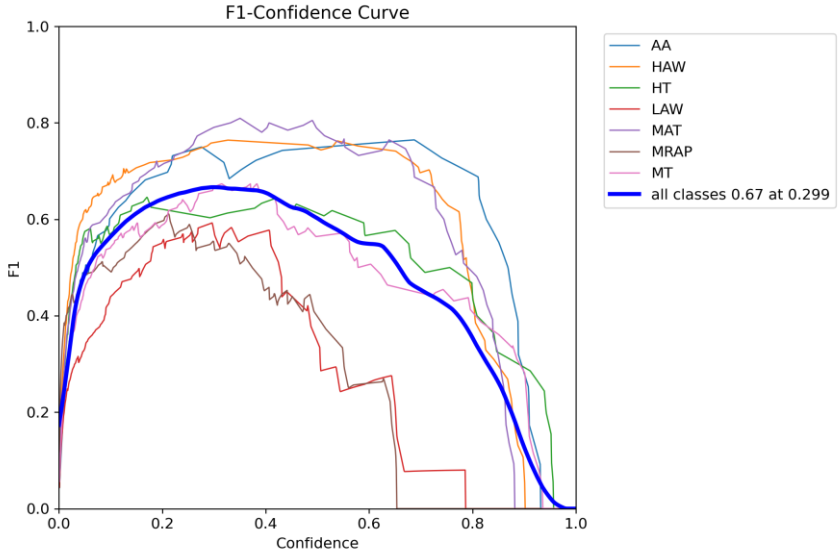

Figure 22. F1-Score of model 11

Instead of analyzing precision and recall over confidence threshold separately, there is another metric named F1-Score [30] that provides a harmonic mean between precision and recall and a balanced assessment of a model's performance. We conclude that model 9 has a better all-round performance.

Mode#	Optimizer	Batch_Size	Cosine_LR	Training_Time_in_minutes	metrics/precision(B)	metrics/recall(B)	metrics/mAP50(B)	metrics/mAP50-95(B)
0	Adam	16	True	84.32226	0.43126	0.58502	0.48069	0.26558

1	Adam	32	True	70.05258	0.43371	0.62472	0.48933	0.28655
2	SGD	16	True	79.8836166	0.57512	**0.64591**	0.63109	0.39577
3	SGD	32	True	67.34346	0.58978	0.63614	0.59388	0.36653
4	Adam	16	False	77,8530624	0.46289	0.57576	0.50294	0.28085
5	Adam	32	False	**65.8506092**	0.44578	0.56925	0.51311	0.30283
6	SGD	16	False	75.4902006	0.57841	0.60231	0.59328	0.37098
7	SGD	32	False	66.938502	**0.63629**	0.61762	**0.64229**	**0.39850**

Table 2. Results of YOLOv5 nano training

As for training YOLOv5 models, there is model 7 that stands out among the others. Models 2 and 3 are also decent candidates, though model 7 is slightly better.

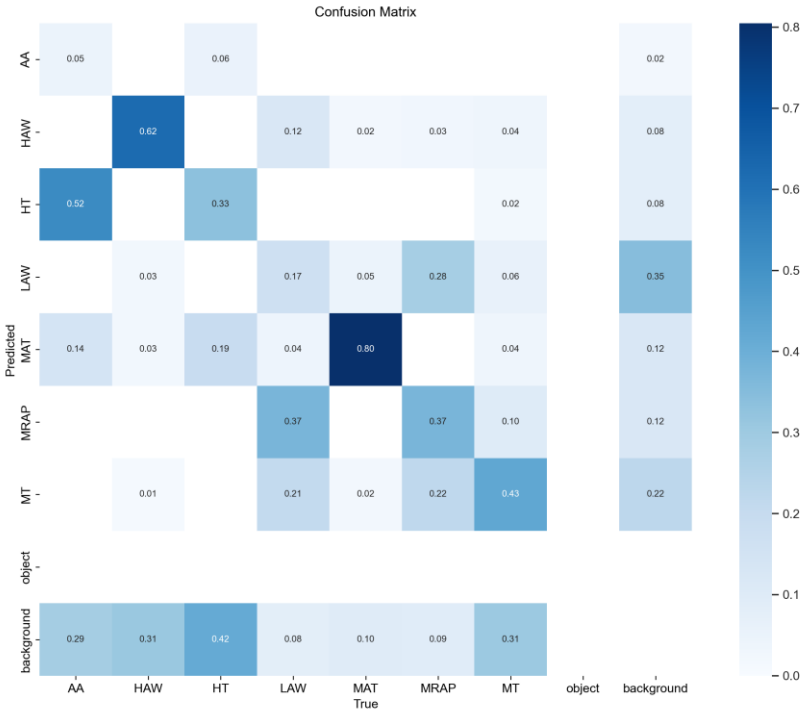

Figure 23. Normalized confusion matrix of model 2

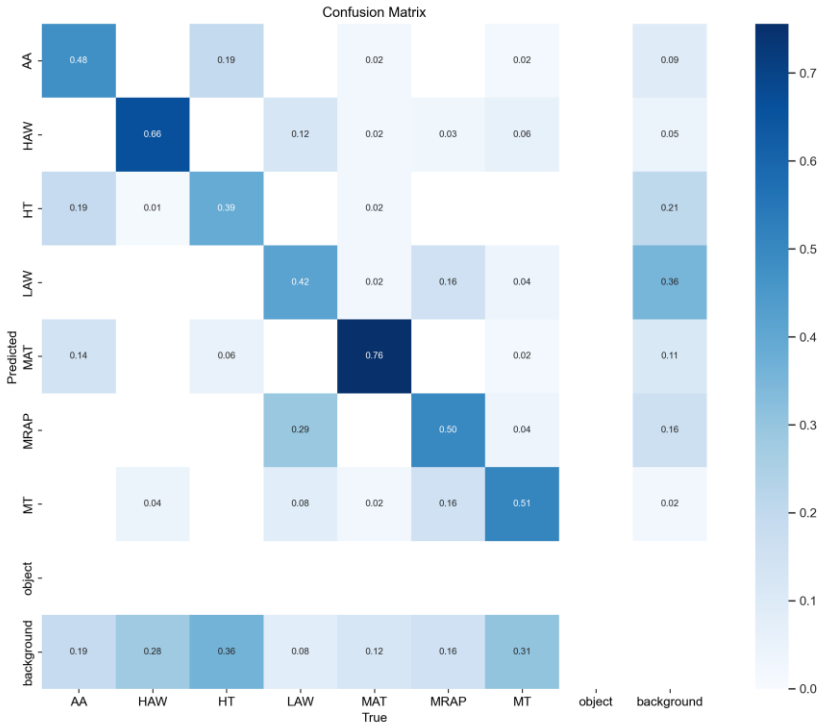

Figure 24. Normalized confusion matrix of model 3

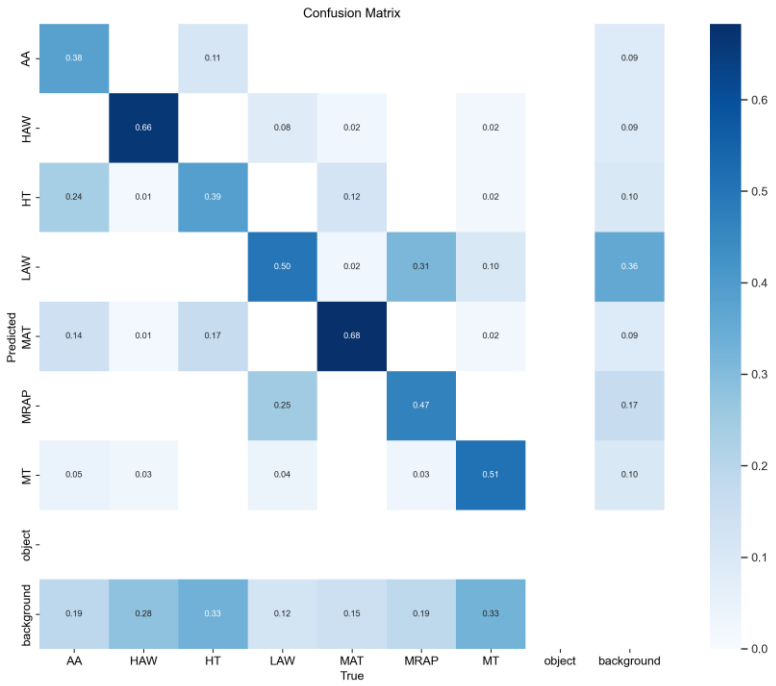

Figure 25. Normalized confusion matrix of model 7

As Figures 23, 24 and figure 25 show that Model 2 not that good performance dealing with many Types I and II Errors, while model 3 and model 7 are similar in the outcome.

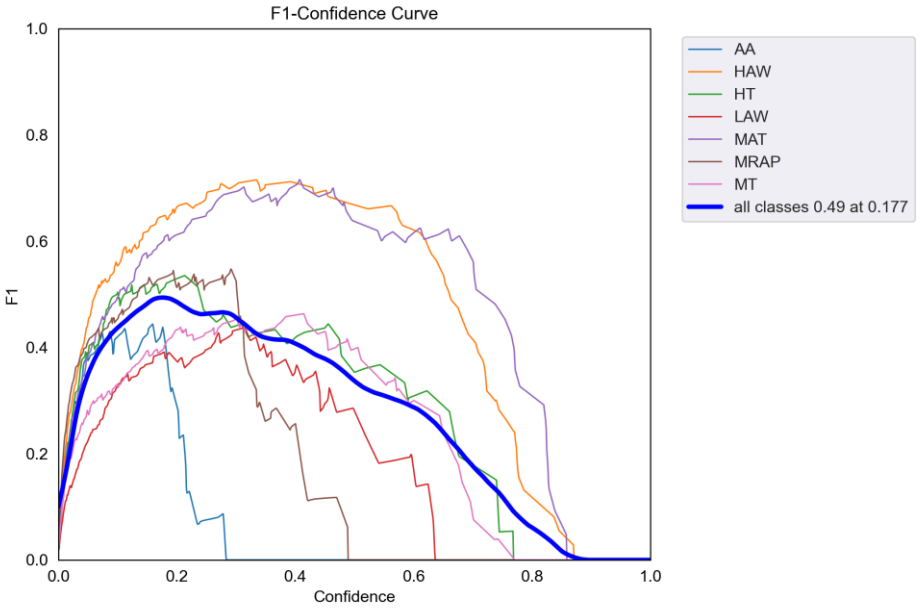

Figure 26. F1-Score of model 2

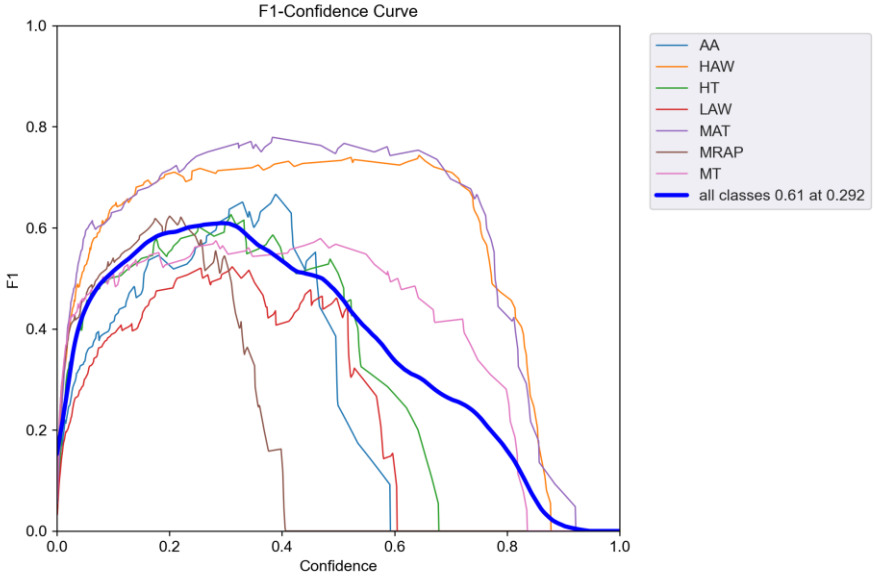

Figure 27. F1-Score of model 3

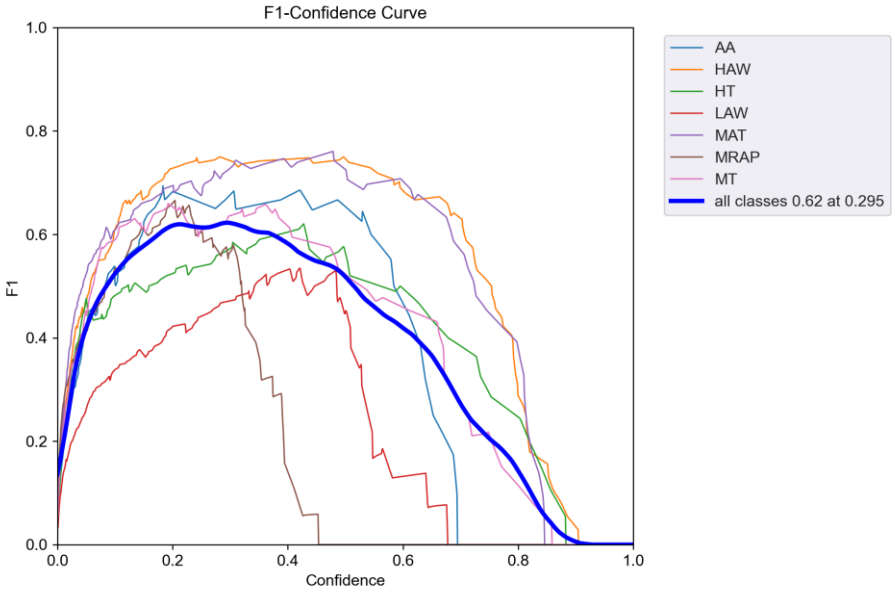

Figure 28. F1-Score of model 7

Figures 26-28 are there to confirm that model 7 in the end is the model with the highest accomplishment among models 2 and 3.

Model	Optimizer	Batch_Size	Cosine_LR	Dropout	Training_Time_in_minutes	metrics/precision(B)	metrics/recall(B)	metrics/mAP50(B)	metrics/mAP50-95(B)	Parameters (in Millions)	FLOPS (in Billions)	Integration of the model into Jetson Nano
YOLOv8 #9	SGD	16	False	0.25	**61.04646**	**0.70702**	0.60424	**0.66581**	**0.44564**	~2.2	7.7	✓
YOLOv5 #7	SGD	32	False	0.00	66.938502	0.63629	**0.61762**	0.64229	0.3985	~1.9	4.5	✓

Table 3. Summary of the training

As Nvidia Jetson Nano possesses a performance for 472 GFLOPS [38] (Floating Operation per Second), the trained YOLOv8 and YOLOv5 produce 7.7 GFLOPS and 4.5 GFLOPS, respectively, we can assume that our models surely can be integrated into Jetson Nano [39] for object detecting in drones, and also in some other, more budget-oriented hardware.

6 CONCLUSIONS

One of the goals of the study was to explore deep learning models that perform real-time object detection of military vehicles. We analyzed a group of YOLO family models, trained them with a variety of parameters and optimizers, and analyzed our training samples of models. We chose the best candidate models among them which can be further integrated in Jetson Nano or other graphical acceleration hardware for accomplishing specific tasks.

6.1 Future scope

The task of creating a custom, more vehicle-diverse dataset to train models on remains to be attended to. We plan to experiment with other deep learning models to compare with YOLO models, enlarge the number of trained models to potentially select an even better architecture with better performance speed trade-off. We have not integrated the trained models into Nvidia Jetson Nano yet or experiment with deployment of the hardware into UAVs, which is an important part of approbation of our method and is left to future work.

REFERENCES

[1] Wikipedia. (n.d.). Unmanned aerial vehicle. Retrieved from https://en.wikipedia.org/wiki/Unmanned_aerial_vehicle#cite_note-:0-2

[2] Commercial drones on the frontline

[3] NVIDIA. (n.d.). NVIDIA Jetson Nano. Retrieved from https://www.nvidia.com/en-in/autonomous-machines/embedded-systems/jetson-nano/product-development/

[4] Wang, G., Chen, Y., An, P., Hong, H., Hu, J., & Huang, T. (2023). UAV-YOLOv8: A Small-Object-Detection Model Based on Improved YOLOv8 for UAV Aerial Photography Scenarios. Sensors, 23(23), 7190. https://doi.org/10.3390/s23167190

[5] Zhai, X., Huang, Z., Li, T., Liu, H., & Wang, S. (2023). YOLO-Drone: An Optimized YOLOv8 Network for Tiny UAV Object Detection. Electronics, 12(12), 3664. https://doi.org/10.3390/electronics12173664

[6] Li, Y., Fan, Q., Huang, H., Han, Z., & Gu, Q. (2023). A Modified YOLOv8 Detection Network for UAV Aerial Image Recognition. Drones, 7(7), 304. https://doi.org/10.3390/drones7050304

[7] Long, J., Shelhamer, E., & Darrell, T. (2015). Fully Convolutional Networks for Semantic Segmentation. arXiv preprint arXiv:1511.08458. https://arxiv.org/abs/1511.08458

[8] DataCamp. (n.d.). Loss Function in Machine Learning. Retrieved from https://www.datacamp.com/community/tutorials/loss-functions-machine-learning

[9] Rosebrock, A. (2016, September 12). Softmax Classifiers Explained. PyImageSearch. Retrieved from https://pyimagesearch.com/2016/09/12/softmax-classifiers-explained/

[10] Girshick, R., Donahue, J., Darrell, T., & Malik, J. (2013). Rich feature hierarchies for accurate object detection and semantic segmentation. arXiv preprint arXiv:1311.2524. https://arxiv.org/abs/1311.2524

[11] Girshick, R. (2015). Fast R-CNN. arXiv preprint arXiv:1504.08083. https://arxiv.org/abs/1504.08083

[12] Ren, S., He, K., Girshick, R., & Sun, J. (2015). Faster R-CNN. arXiv preprint arXiv:1506.01497. https://arxiv.org/abs/1506.01497

[13] Redmon, J., & Divvala, S. (2015). You only look once: Unified, real-time object detection. arXiv preprint arXiv:1506.02640. https://arxiv.org/abs/1506.02640

[14] Redmon, J., & Farhadi, A. (2016). YOLO9000: Better, faster, stronger. arXiv preprint arXiv:1612.08242. https://arxiv.org/abs/1612.08242

[15] Redmon, J., & Farhadi, A. (2018). YOLOv3: An incremental improvement. arXiv preprint arXiv:1804.02767. https://arxiv.org/abs/1804.02767

[16] Bochkovskiy, A., Wang, C. Y., & Liao, H. Y. M. (2020). YOLOv4: Optimal speed and accuracy of object detection. arXiv preprint: 2004.10934. https://arxiv.org/abs/2004.10934

[17] Ultralytics GitHub Repository. (n.d.). YOLOv5. Retrieved from https://github.com/ultralytics/yolov5

[18] PyTorch. (n.d.). PyTorch. Retrieved from https://pytorch.org/

[19] Ultralytics GitHub Repository. (n.d.). YOLOv8. Retrieved from https://github.com/ultralytics/ultralytics

[20] Tan, M., Pang, R., & Le, Q. V. (2024). YOLOv9. arXiv preprint arXiv:2402.13616. Retrieved from https://arxiv.org/abs/2402.13616

[21] YOLOv5 architecture image. Retrieved from https://www.youtube.com/watch?v=LwtxRy26pVY&t=2386s&ab_channel=HelloPaperspace

[22] Ioffe, S., & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. arXiv preprint arXiv:1502.03167. Retrieved from https://arxiv.org/abs/1502.03167

[23] PyTorch Documentation. (n.d.). torch.nn.SiLU. Retrieved from https://pytorch.org/docs/stable/generated/torch.nn.SiLU.html

[24] Ultralytics GitHub Issue Tracker. (n.d.). BottleNeckCSP Layer. Retrieved from https://github.com/ultralytics/yolov5/issues/12677

[25] Medium. (n.d.). Unsampling, Unpooling, and Transpose Convolution. Retrieved from https://medium.com/jun94-devpblog/dl-12-unsampling-unpooling-and-transpose-convolution-831dc53687ce

[26] RangeKings GitHub Repository. (n.d.). YOLOv8 architecture. Retrieved from https://www.researchgate.net/figure/YOLOv8-architecture-from-RangeKings-GitHub-repository-showcasing-its-innovative-design_fig2_376357696

[27] - Ultralytics GitHub Repository. (n.d.). Issue #189. Retrieved from https://github.com/ultralytics/ultralytics/issues/189

[28] - Rakhecha, A. (n.d.). Understanding Learning Rate. Towards Data Science. Retrieved from https://towardsdatascience.com/https-medium-com-dashingaditya-rakhecha-understanding-learning-rate-dd5da26bb6de

[29] - Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980. Retrieved from https://arxiv.org/pdf/1609.04747.pdf

[30] - Ultralytics. (n.d.). YOLO Performance Metrics. Retrieved from https://docs.ultralytics.com/guides/yolo-performance-metrics/

[31] - Kaggle. (n.d.). Retrieved from https://www.kaggle.com/

[32] - Roboflow Universe. (n.d.). Retrieved from https://universe.roboflow.com/

[33] - Roboflow Universe. (n.d.). Baskent Univercity mcv2 dataset. Retrieved from https://universe.roboflow.com/baskent-univercity-5cybx/mcv2/dataset/1

[34] - Ultralytics. (n.d.). COCO Dataset. Retrieved from https://docs.ultralytics.com/datasets/detect/coco/

[35] - Tan, M., & Le, Q. V. (2019). EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. arXiv preprint arXiv:1905.11946. Retrieved from https://arxiv.org/pdf/1905.11946.pdf

[36] - Ultralytics GitHub Repository. (n.d.). Issue #7304. Retrieved from https://github.com/ultralytics/ultralytics/issues/7304

[37] - Ultralytics GitHub Repository. (n.d.). Issue #5517. Retrieved from https://github.com/ultralytics/yolov5/issues/5517

[38] - ResearchGate. (n.d.). Parameter count and FLOP comparison. Retrieved from https://www.researchgate.net/figure/Parameter-count-and-FLOP-comparison-Params-represent-the-number-of-model-parameters_fig6_362843939#:~:text=Params%20represent%20the%20number%20of%20model%20parameters.,operations%20required%20by%20the%20model.&text=Person%20re%2Didentification%20is%20essential,as%20behavior%20and%20event%20analysis

[39] - Ultralytics. (n.d.). NVIDIA Jetson. Retrieved from https://docs.ultralytics.com/guides/nvidia-jetson/#what-is-nvidia-jetson

APPENDIX 1

In [ ]:

```python
def train_yolo_model(optimizer, batch_size, cos_lr, dropout, epochs, imgsz):
    start_time = time.time()

    model = YOLO("C:/Users/Cyberpower/PycharmProjects/modelTraining/models/custom_yolov8n.yaml")

    model.train(data="C:/Users/Cyberpower/PycharmProjects/modelTraining/data_yolov8/data.yaml",
                epochs=epochs,
                imgsz=imgsz,
                device=0,
                optimizer=optimizer,
                dropout=dropout,
                cos_lr=cos_lr,
                batch=batch_size)

    duration = time.time() - start_time

    print(duration)

    return duration


```

In [ ]:

```python
df = pd.DataFrame(columns=['Optimizer', 'Batch_Size', 'Cosine_LR', 'Dropout', 'Training_Time_in_seconds', 'Training_

# Hyperparameters

epochs = 100
imgsz = 224
optimizers = ['Adam', 'SGD']
batch_sizes = [16, 32]
cos_lr = [False, True]
dropouts = [0.0, 0.25]

for optimizer in optimizers:
    for batch_size in batch_sizes:
        for use_cos_lr in cos_lr:
            for dropout in dropouts:
                duration = train_yolo_model(optimizer, batch_size, use_cos_lr, dropout, epochs, imgsz)

                df = df.append({'Optimizer': optimizer,
                                'Batch_Size': batch_size,
                                'Cosine_LR': use_cos_lr,
                                'Dropout': dropout,
                                'Training_Time_in_seconds': duration,
                                'Training_Time_in_hours': duration/3600},
                                ignore_index=True)
                time.sleep(1200)
```

Figure 13. Training's implementation on custom YOLOv8 nano